From (Imperfect) Object Diagrams to (Imperfect) Class Diagrams

New Ideas and Vision Paper

Andreas Kästner
University of Bremen
Bremen, Germany
andreask@cs.uni-bremen.de

Martin Gogolla
University of Bremen
Bremen, Germany
gogolla@cs.uni-bremen.de

Bran Selic
Malina Software Corp.
Ottawa, Canada
selic@acm.org

ABSTRACT

In order to achieve effective support for software development, the transition between an informal and provisional mode of tool operation, which is conducive to design exploration, and a formal mechanistic mode required for computer-based design capture is crucial. This contribution proposes a smooth transition for designing class models starting from informal, sketchy object models. We propose a lenient development approach and discuss the possibilities and problems of a transformation from object diagrams to class diagrams. While classes describe abstract concepts, objects are representations of what can be seen in the real world, so it might be easier to start modeling with objects instead of classes. An object diagram can however not describe a whole system, it is only used as the first step of an iterative process to create a complete model. During this process, our object and class diagrams provide a notation for highlighting missing or conflicting parts. Based on these imperfect object diagrams, educated guesses can be made for resulting, imperfect class diagrams, which can then be refined to a complete, formal description of the modeled system.

KEYWORDS

UML object diagram, UML class diagram, Incremental transformation by example, Tool support

ACM Reference Format:
Andreas Kästner, Martin Gogolla, and Bran Selic. 2018. From (Imperfect) Object Diagrams to (Imperfect) Class Diagrams: New Ideas and Vision Paper. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3239372.3239381

1 INTRODUCTION

Increased levels of computer-supported automation are considered as one of the key enablers to the higher levels of productivity and product quality promised by Model-Based Engineering (MBE). However, practical experience with present-day MBE tools indicates that we are still far from this ideal. Often, tools are complex, difficult to learn, and difficult to use. Users often find themselves in frustrating situations where the tools are forcing them into workarounds and constrained operating modes which are not conducive to free expression of ideas. In order to achieve effective support from tools, the transition between an informal and provisional mode of tool operation, which is conducive to design exploration, and a formal mechanistic mode required for computer-based design capture is crucial.

This contribution is intended to be one step among many others in the development of tools that support users in constructing models through freely formulating ideas. We aim at a smooth transition for designing class models starting from informal, sketchy object models. We propose a lenient development approach and discuss the possibilities and problems of a transformation from object diagrams to class diagrams. While classes describe abstract concepts, objects are representations of what can be seen in the real world, so it might be easier to start modeling with objects instead of classes [1]. Therefore, "Objects Before Classes" is an approach that comes with good arguments [19]: our basic idea is to create (imperfect) class diagrams based on imperfect object diagrams.

> "It is logically impossible to induce the general case from a set of examples, but well-chosen prototypes are the way most people think." - Rumbaugh et al. [14, p. 19]

The syntax of our proposed concepts for object and class diagrams is based on UML. Because of their exemplary nature, object diagrams are not able to completely describe a system [15, p. 140]. Furthermore, when creating object diagrams, there are multiple possibilities, how inconsistencies or formal mistakes can happen. An example would be when different people work on different parts, and they use slightly different names for the same modeled concept. At other times, modelers just want to write down their ideas while some parts, for example exact names, are not yet clear. For this reason, our approach is a lenient one that supports to formulate informal object diagrams. During the development process, both object and class diagrams do not have to follow accurate UML syntax. To highlight the informal parts, the terms "Incomplete" and "Contradictory" are used. "Incomplete" means that parts can be missing. For example, it should be allowed that there is an attribute without a name. "Contradictory" means that inconsistencies should be allowed. For example, it should be possible for two objects of the same class to have an attribute with the same name, but with different data types. The resulting class diagram highlights "Incomplete" and "Contradictory" elements so that they may be fixed in

MODELS '18, October 14–19, 2018, Copenhagen, Denmark

ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239381

the diagram. This way, an iterative process is supported where the diagrams can become complete and formal UML diagrams step by step. This lenient approach could also be a way to reduce the "massive productivity loss"[13] of model developers who are not modeling experts and only model occasionally.

The research contribution of this paper is to close the gap between the assumed natural object-oriented thinking of developers and formal class-oriented modeling. A transformation is developed from informal (or imperfect) object diagrams to also informal (or imperfect) class diagrams. The transformation is implemented as a plug-in for the tool USE [6, 7], and is evaluated through an experiment with developers familiar with object-oriented concepts. The contribution in [8] shortly mentions an example transformation, but the current paper gives a comprehensive overview on the approach.

The structure of the rest of this paper is as follows. Section 2 gives the general idea of our approach and introduces the transformation through an example. Section 3 explains the metamodels on the source and target sides of the transformation. Section 4 connects these metamodels by elaborating the transformation in detail. To validate our approach, an evaluation with developers familar with modeling was undertaken, and the results are presented in Sect. 5. Related work is discussed in Sect. 6. Finally, a conclusion, as well as discussion for future work, is given in Sect. 7.

2 BASIC IDEA AND RUNNING EXAMPLE

2.1 Extensions to Object and Class Diagrams

UML object and class diagrams were designed in order to *precisely* describe single system snapshots and the general structure of a system. They were not developed having in mind a *flexible* way of developing structures. To get closer to a soft development process and to support the leniency that is the aim of our approach, extensions had to be made to the usual UML object and class diagrams. Leniency allows developers to sketch out rough ideas and analyze them, usually in the presence of others, to determine if they are worth developing further or whether they should be discarded because they do not seem promising. It is important to detect bad designs early, since (a) it reduces the overall amount of effort and, (b) from a psychological point of view, developers are generally more prone to discard rough ideas than ideas into which they have invested much time and effort.

In our approach, an object diagram should embrace the exemplariness of typical, even partial snapshots and allow for exploration, mistakes, and imperfection during the development process. Therefore, an object diagram does not have to follow the exact UML syntax, as shown in Fig. 1. It is allowed to have inconsistencies as well as to leave parts open that are usually required by UML tools. But feedback has to be given to the developers about the difference between the status of their actual object diagram and the formal expectation from UML. Incomplete or inconsistent parts should be highlighted during development. This is where the concept of the markers (<+> , <?> , <!>) extend the usual object and class diagram features. <+> is only used in object diagrams and signifies a spot where additional information can still optionally be added. <+> is shown instead of an actual value whenever a new object or link is created. <?> and <!> are only used in the class diagram and show up, when the transformation (from object to class diagram) cannot induce complete information about the classes or associations. While <?> signifies missing information, <!> is used to highlight conflicts that can occur through inconsistent developer specification, in technical terms it arises by merging different objects or links. The detected problematic modeling elements (like the data type of an attribute or a role name) are additionally highlighted during the development process in both diagram types within dashed rectangles or with dashed lines in order to indicate that these elements are not complete and that more developer action is needed.

2.2 Running Example Details

Figure 1 shows an incomplete and inconsistent object diagram, and Fig. 2 presents the induced imperfect class diagram, as presented and deduced in the tool USE. The example is based on a Project-Department-Employee context. From the objects of class `Project` and the links in association `WorksIn`, one can determine a complete class and a complete association specification, where, for the multiplicities, narrow intervals deducible from the links are chosen. Also, the role names for `WorksIn` have been stated in a consistent way. 'Perfect' elements are presented with a solid, non-dashed class rectangle or a solid, non-dashed association line. All other class diagram elements are presented with dashed rectangles and lines indicating imperfection in the object diagram. The class `Employee` is considered as being incomplete because the object `james` in the object diagram is missing an attribute identifier. Probably it was meant to be an attribute `name`, however, that cannot be automatically determined. Instead, in the class diagram, the attribute is listed as an attribute with an unspecified identifier of type `String`. As a further step during the iterative development process, a developer could fix the omission in the source object diagram. The association `Controls` is classified as incomplete because a role name on the `Department` side has never been explicitly mentioned in the object diagram. The class `Department` is considered contradictory because the value of the attribute `budget` is specified as a `String` in one object and as an `Integer` in another object. Thus `budget` is highlighted by the contradictory marker <!>. Finally, the association `WorksOn` is considered contradictory, because in one link the role name at the `Project` side is given as `project` and in another link as `PROJEKT`. This is an obvious contradiction in this case, but our functionality also detects more subtle inconsistent role names that may occur in real-life modeling scenarios. The following table gives a short overview in which diagram which marker applies together with the meaning of the marker.

Marker	<+>	<?>	<!>
Object diagram	can be extended	-	-
Class diagram	-	missing info	contradictory

3 UNDERLYING METAMODELS

To get a better understanding of the transformation, this section explains its source and its target through the use of metamodels.

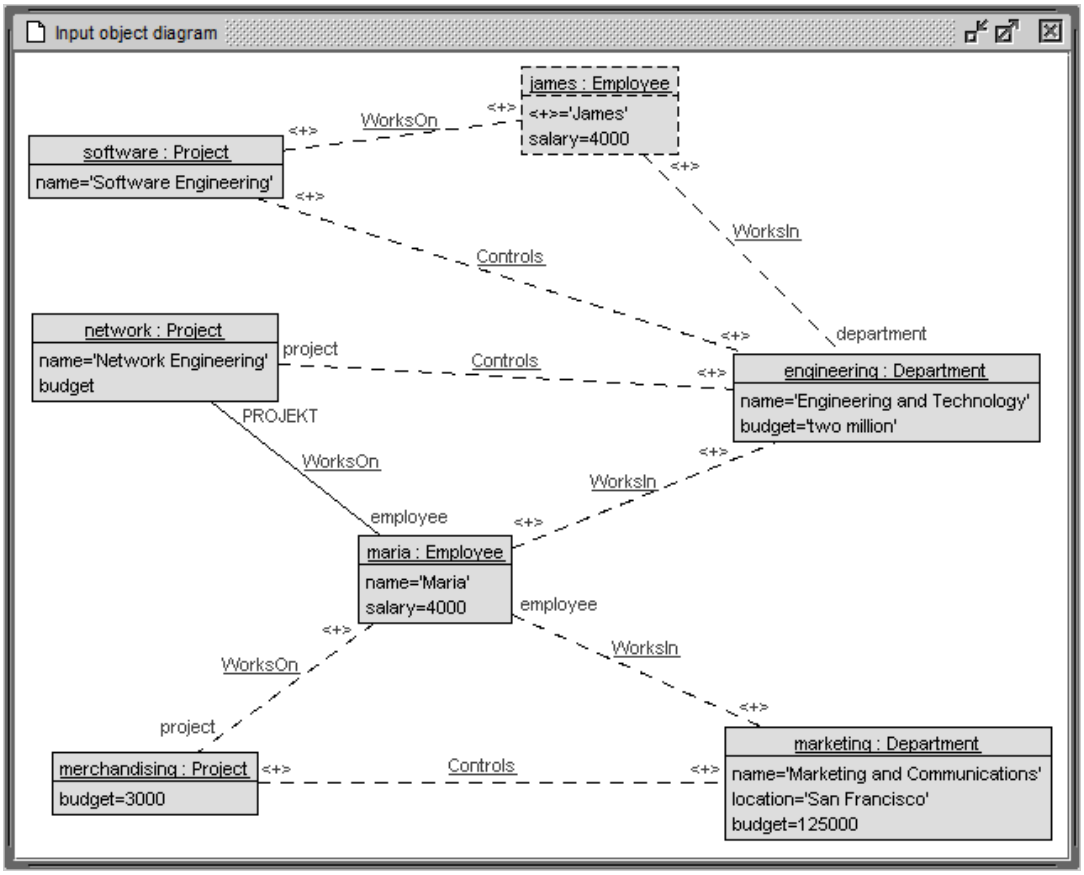

Figure 1: An example for an imperfect object diagram. The dashed lines and markers highlight the imperfect parts.

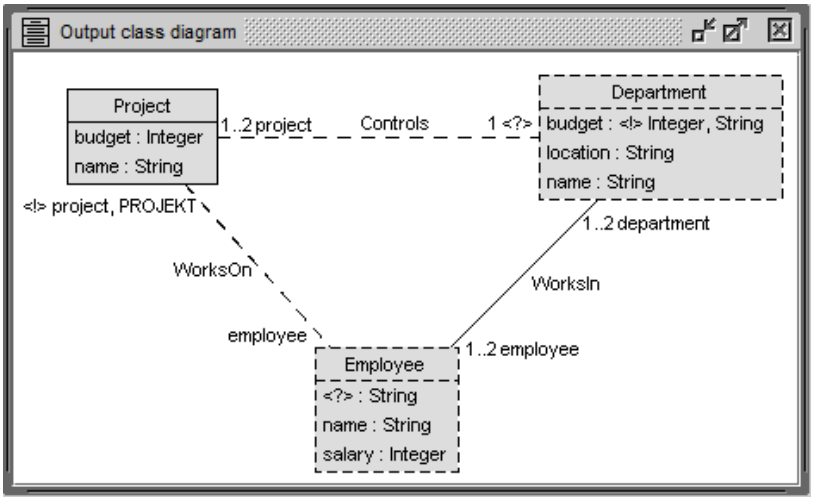

Figure 2: An imperfect class diagram. The result of transforming the imperfect object diagram in Fig. 1.

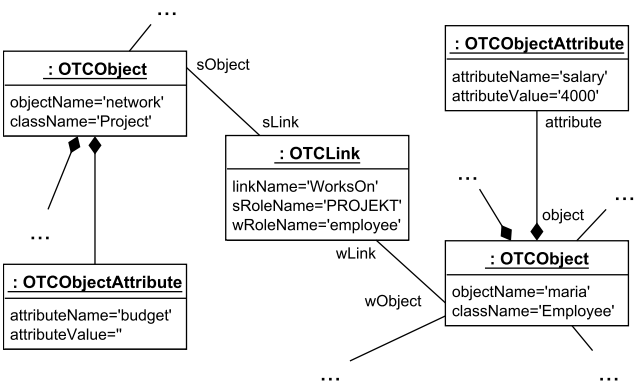

Figure 3: Internal metamodel view of a part of Fig. 1

3.1 Source Metamodel

Following the "Objects Before Classes" approach of this paper, we start to introduce the metamodel by giving an example of its instantiation. Figure 3 shows the part of Fig. 1, where the object `maria` and the object `network` are connected by the link `WorksOn`. To keep the example simple, only a few objects are shown. The rest of the existing diagram is only implied. The `OTCLink` (OTC=Object to class) is connected to exactly two `OTCObjects` because it represents a binary association. Since the two ends of the association should have no specific order, but nonetheless must be distinguishable, the two prefixes `s` and `w` are introduced. One may think of them as standing for "summer" and "winter" if that is easier to remember, but any other arbitrary contrastive pair would work as well. So for example, the attribute `sRoleName='PROJEKT'` belongs to the side of the link with the role names `sLink` and `sObject` because they all have the prefix `s`.

Figure 4 shows the complete metamodel of the object side. Each attribute is of the type `String`, since it represents a value that can directly be typed in by the user. Each of the three classes has a function `getStatus()` that returns one of two values. If it returns `Complete`, all information needed is available. If it returns `Incomplete`, there is still information missing which gets graphically represented by <+>. The associations describe the relations between the `OTCObject` and the other parts.

3.2 Target Metamodel

Figure 5 shows the metamodel of the class side. The role name attributes of the `OTCAssociation` class are sets of strings that get created by merging role names from the different links specified by the user. In our running example, one role name attribute would have the value {project, PROJEKT}. Since this set has more than one element, the `getStatus()` function would return `Contradictory`. The attribute `attributeTypes` of the class `OTCClassAttribute` is also a set that gets created in a similar way. In our running example, one attribute `attributeTypes` would have the value {String, Integer}. Again, as the set has more than one element, the status would be `Contradictory`.

Currently supported and recognized are the five data types listed in `OTCType`. The status can be one of three values. `Incomplete`

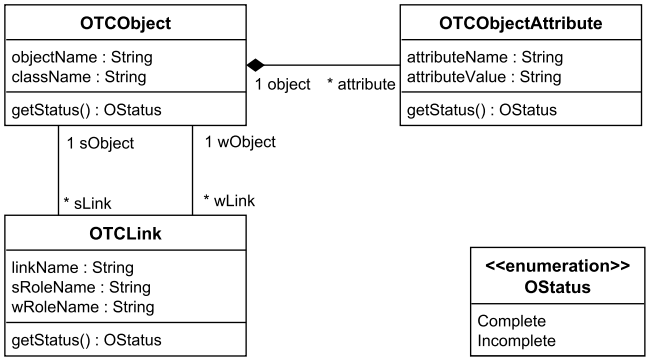

Figure 4: Underlying metamodel on the object side.

means there is missing information which is graphically represented by <?>. `Contradictory` means there is conflicting information which is graphically represented by <!>. Otherwise the status is `Complete`. The multiplicities have their own data type internally, but to keep the metamodel simple, they are represented as `String` in this figure.

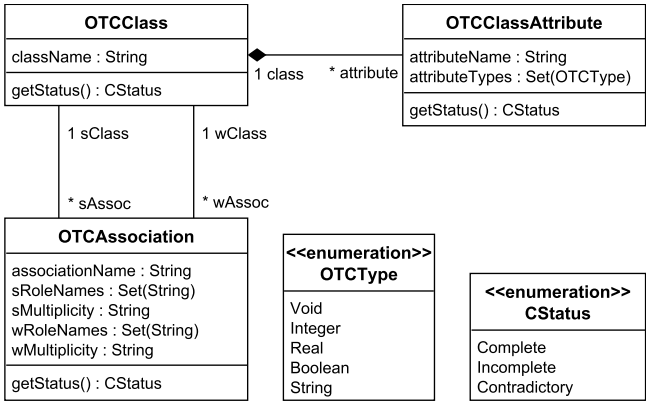

Figure 5: Underlying metamodel on the class side.

4 DETAILS OF THE TRANSFORMATION

After introducing source and target in the last section, now the actual details of the transformation are explained.

The transformation is divided into two main parts. First, the classes are created based on the objects (later called object-to-class or OTC). Then, the associations are created based on the links (later called link-to-association or LTA). This order cannot be reversed because the second step needs information about which objects are mapped to which classes. Figure 6 shows a high-level view of the transformation process. The rest of this section explains each step shown in that diagram.

4.1 From Objects to Classes

The creation of classes is further divided into two steps. First, the actual classes are created. The second step handles the attributes inside each class and solves possible conflicts.

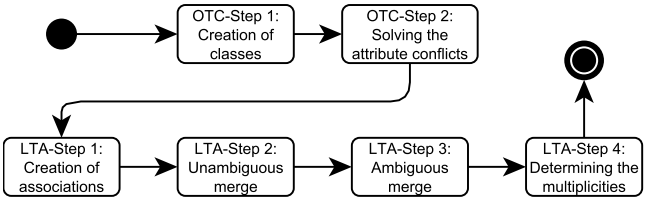

Figure 6: Overview of the six steps of the transformation.

4.1.1 OTC-Step 1: Creation of classes based on the objects. For every object in the object diagram, there are two possible outcomes: creating a new class or merging information into an existing one (see Fig. 7). If the current object has a new class name that was not handled before, a new class will be created. That includes objects without a class name. If a currently handled object has a class name, for which a class already exists, it gets merged into that class. What that means, is that the attributes of the current object are simply added to the temporary attributes of that class. After every object in the object diagram was handled this way, all expected classes are present. However, it is possible that there is a conflict between attributes within a class. These kinds of conflicts are solved within the next step.

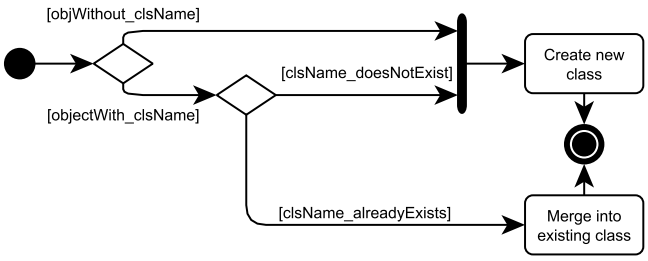

Figure 7: Decisions and outcomes for every object.

4.1.2 OTC-Step 2: Solving the attribute conflicts. Now the classes exist, but their temporary attributes are just directly copied from the objects. To create the definite attributes, each temporary attribute is handled one by one, as shown in Fig. 8. The four paths leading up to the final state, are now explained.

"Create new attribute" path: If the attribute has no name or the name does not exist already in the definite attributes, the current attribute just gets added to the definite attributes. That is because the information that comes with this attribute cannot be integrated anywhere else.

"Do nothing" path: If the name exists, but the corresponding value is either empty or of the same type as the existing attribute, the information just gets ignored. The information about the attribute is already there, so it does not have to be added.

"Adopt type" path: If the name of the current attribute already exists, but the type was previously not known, the type of the current attribute gets adopted. During this path, the additional information gets added to the already existing name. Since there was no type known before, the resulting attribute is complete.

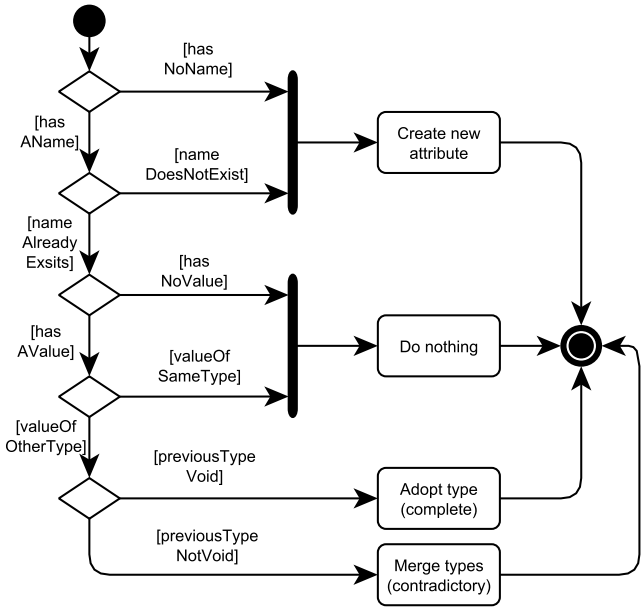

Figure 8: Decisions and outcomes for every temporary attribute in a class.

"Merge types" path: If the name of the current attribute already exists, but the type was previously different, the type gets merged with the previous one. During this path, previous information directly conflicts with current information. The resulting attribute is therefore contradictory and marked with a <!>.

4.2 From Links to Associations

The creation of associations is further divided into four steps. First, the associations are created by adopting the links. Secondly, the associations that can be unambiguously merged, will be merged. Thirdly, associations that might belong together get merged ambiguously. Finally, the multiplicities are determined.

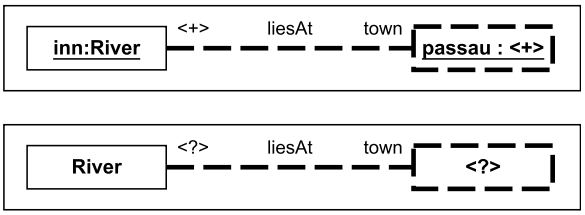

Figure 9: Link adoption example.

4.2.1 LTA-Step 1: Adopting the Links. First of all, for every link that was put into the transformation, an association is created. An example is given in Fig. 9. There are five items, that need to be considered: the two adjacent objects, the link name, and the two role names. The three mentioned names that may or may not be empty are just copied. To find the corresponding classes, the information that was gathered during the transformation of the objects is used. It is not possible to just use the class name to identify the correct

class because it can also be empty. In such a case, the association should be connected with the exact class, that was created from the adjacent object of the current link.

4.2.2 LTA-Step 2: Unambiguously merging the associations. At this point, there are as many associations as there were links, they are already between the correct classes, but their number has to be reduced. During this step, every newly created association is compared to other associations, to see if they can be unambiguously merged. The idea behind the unambiguous merge is that two associations get merged in a way, that does not leave room for speculation. For this reason, there are several rules for such a merge.

General rules. It is necessary that both possible merge candidates have the same adjacent classes. Then, the association names also have to be the same. From there, the criteria for that kind of merge are different for reflexive and non-reflexive associations.

Non-reflexive associations. If the general rules for a merge apply, it always happens. The only thing that has to be considered, is the correct order of the merge. But because of the different adjacent classes, the role names can be clearly assigned to one side, even if some of them are missing. The result for this case can be seen in Fig. 10. As shown by the curly brackets, the resulting role names are stored as a set. If the set contains more than one role name, the association becomes contradictory. If the set is empty, the association becomes incomplete. Note that the names of the objects have no influence on the target diagram and are just allowed for convenience [11].

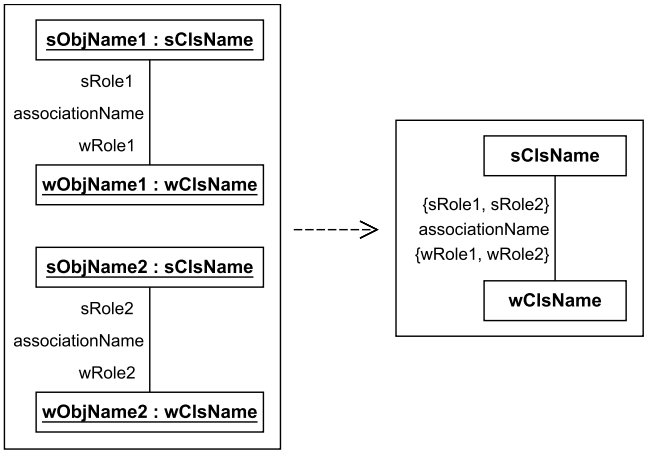

Figure 10: Unambiguous merge for non-reflexive associations.

Reflexive associations. In addition to the general rules, there is also another criterion that needs to be fulfilled for the merge to happen. One association has to have both role names. What could happen if the rules were less strict, is shown in Fig. 11. Because both of the links are missing a second role name, the order of the merge is not clear. This leads to a merge of two semantically different associations. This behavior is not wanted during this unambiguous step. However, it should be allowed during the next step. Of course, it is also possible to misspell the association name. That would, however, lead to two different associations, which should be easily noticeable in the class diagram.

These conditions are made in a way, that demands a lot from the input. However, inconsistencies are guaranteed to be found. That is why the line between merging and not merging was drawn at these exact places.

4.2.3 LTA-Step 3: Ambiguously merging the associations. To allow for more leniency during the development process, the creators of the object diagram should be given freedom to leave parts empty. This is why a second merge happens. If two associations get merged here, they always result in an incomplete association to signify that assumptions were made during the transformation. The basic idea of this step is to merge everything, as long as it is not contradictory.

During this step, the complete and incomplete associations from Step 2 are merged again, but this time with less strict rules. The contradictory associations are excluded because they already have conflicts, which need to be solved. The result will be a mix of complete associations (if no merge has been done for one input association) and incomplete associations (if at least one merge has been done).

This time, reflexive and non-reflexive associations have the same requirements for a merge. The order of roles in reflexive associations is guessed, if necessary. The only two requirements to merge two associations are, that they are between the same classes and that no role or association name contradicts each other. Note that no new contradictory associations can be created during this step. However, there can be semantic errors as shown in Fig. 11. This is why all associations that are merged during this step are incomplete, even when they are semantically correct as shown in Fig. 12.

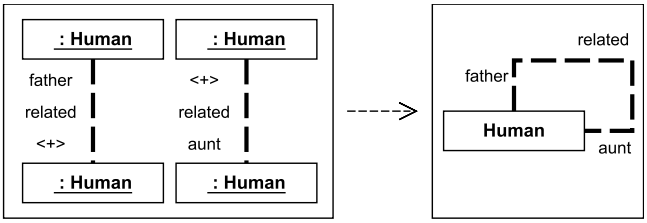

Figure 11: A semantic error during the ambiguous merge. The possibility of a faulty association is highlighted by the dashed line.

4.2.4 LTA-Step 4: Determining the Multiplicities. Calculating the intended multiplicities based on an object diagram is impossible. Booch stated that "When you model your system's design view, a set of class diagrams can be used to completely specify the semantics of your abstractions and their relationships. With object diagrams, however, you cannot completely specify the object structure of your system."[2]. That is because an object diagram always displays a specific state in time of the modeled system. Therefore it cannot describe the whole system at all times. No matter how many links are projected on the same association, the * multiplicity can never be reached. Also, it could always be the intention (even if unlikely)

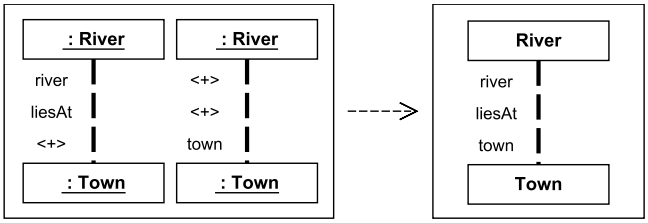

Figure 12: No semantic error was made during the ambiguous merge, but since there is no automatic way to detect this, the association is marked as incomplete.

of the original diagram to model a very high but finite multiplicity like `0..10000`. A simple approach would be to set every multiplicity as `0..*`. That would never be strictly wrong. However, then the transformation might as well not set multiplicities at all since no new information is available. Right now, the idea is to calculate the minimal and maximal multiplicities as they can be obtained from the source diagram. However, the setting of the multiplicities leaves room for improvements. It might be good to have a post-processing phase for the multiplicities in future work, where the multiplicities can be changed manually by users of the transformation.

5 EVALUATION

The concepts and implementation of our approach were evaluated with the help of test subjects in the form of a survey.

5.1 Research Questions

Two goals should be achieved with the research questions. The main goal is to find an answer if the concepts of the transformation hold up to what possible users of the tool think or if there are different approaches that might be considered to be implemented. The secondary goal is to test the tool itself. Is the implementation already in a good enough state, that allows users to work with it to create acceptable results?

Research question 1 Are there any patterns of behavior when the participants try to come up with a transformation from object diagrams to class diagrams?

Research question 2 Do those patterns of behavior match the functionality of the transformation proposed in this paper? How do they differ? If they differ, do the participants agree with the version of the transformation that was proposed in this paper?

Research question 3 Is the tool in its current state advanced enough to be used to deliver acceptable results?

First, it needs to be answered if participants who have no prior experience with the transformation have a common approach to it. The data from the survey needs to be evaluated in a way that shows these similarities.

If the first research question is answered positively, it would be interesting to see, if the patterns, the participants came up with, match the approach of this paper. If there are differences, can any of the new ideas be included in future work for this transformation? It is to be expected, that differences are produced by the participants. However, it is relevant for the validation of the transformation to find out, if participants agree with the proposed version.

The last research question goes in a different direction. Instead of conceptual ideas, the usability of the actual tool should be evaluated. There still need to be improvements in that regard, but it would be interesting to see if potential users are able to work with the current state of the implementation.

5.2 Survey Realization

5.2.1 General Setup. The evaluation was scheduled for and executed on June, 22nd 2017 at the University of Bremen. The evaluation lasted 90 minutes, in which the first 30 minutes were used as an introduction to the topic, as well as for other preparations. The final 60 minutes were used as time for the participants to fill out the survey. This main part consisted of both conceptual tasks as well as working with the developed tool. Every participant received a quickstart guide to the plugin on paper to reduce the possibility to be stuck on simple tasks, which are irrelevant to the research questions. Also distributed on paper were the two prepared examples and a place for drawing[1]. It was decided to use a survey, created with Google Forms, which was available online[2]. Also available online, was the USE version with the plugin and the examples[3].

5.2.2 Participants. In order to be able to participate in the survey, the persons chosen need to have at least a basic understanding of UML. To be more precise, they need to know the basic structure of object and class diagrams. If a test subject does not have this knowledge, it would make little sense to ask them to speculate, how a transformation from objects to classes could look like. Experience with the standard version of USE would be beneficial but not required. In order to find such persons, it was possible to work with the participants of the course "Design of Information Systems"[4]. Furthermore, additional persons with UML experience participated together with the participants of the course. In total, there were 12 participants who have performed the task on paper. However, only 11 of those submitted the online survey.

5.3 Survey Results

The unaltered result data from the survey, except for removing the email addresses, is available online[5].

5.3.1 Research question 1: To answer research question 1, patterns of behavior had to be found for the two examples. The first example was rather simple and the patterns were mostly as expected. On the other hand, the second example was already introduced in this paper (Fig. 1) and it was more complex, so only those results are shown (Fig. 13). To reveal a new pattern, there needed to be at least three people who used the same approach.

Since the research question was merely to look for patterns and patterns were found, the answer to the question is positive.

5.3.2 Research question 2: The general approach that the participants had was very similar to the approach proposed in this paper,

[1] https://drive.google.com/file/d/16728FUl00_Zjaax9ikxePn9a0hjE8GA6
[2] https://goo.gl/forms/6KoMFIGVe5ebooWr1
[3] http://www.db.informatik.uni-bremen.de/publications/intern/use_otc.zip
[4] http://www.db.informatik.uni-bremen.de/teaching/courses/ss2017_eis/
[5] https://goo.gl/hAJkAq

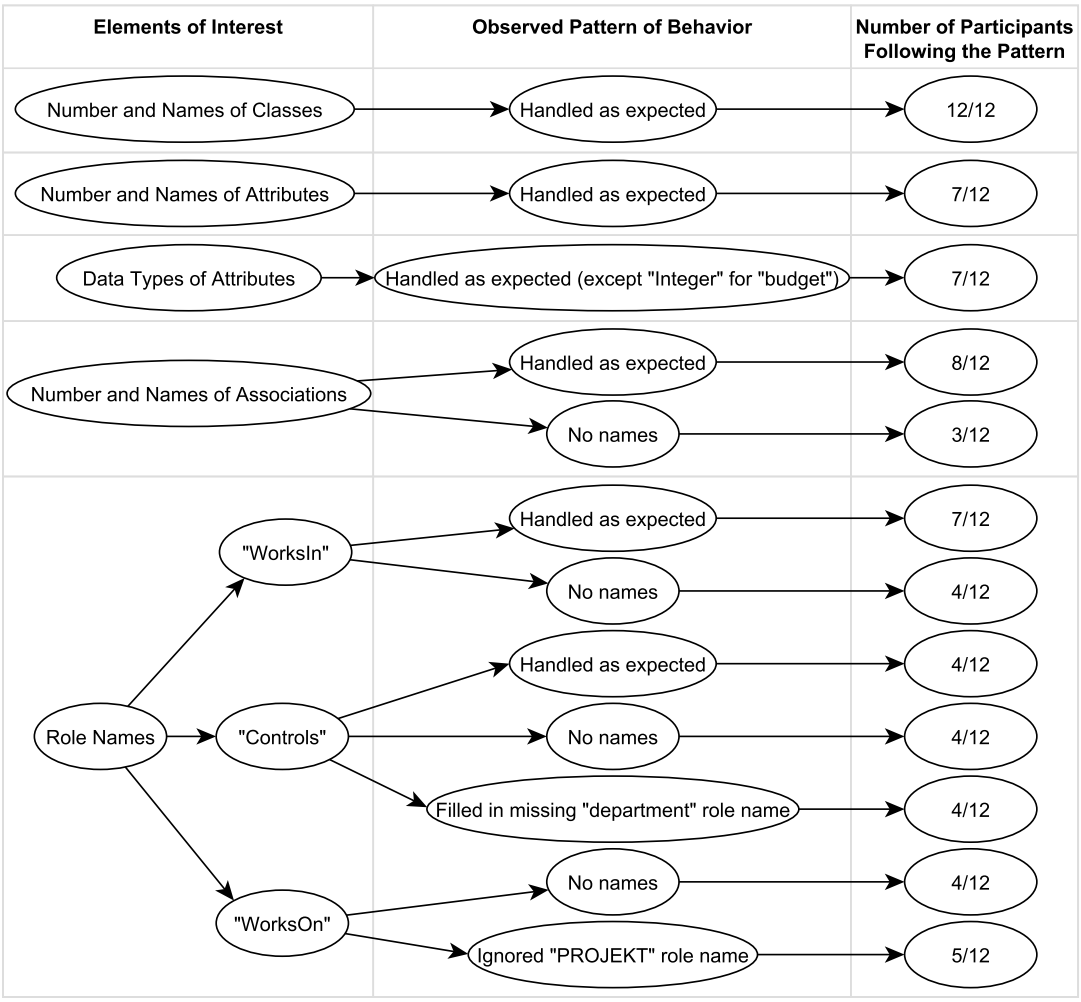

Figure 13: Observed patterns of behavior for the second example. The first column show the elements that were examined and the second column shows the observed patterns for these elements. The third column shows how many participants followed such a pattern.

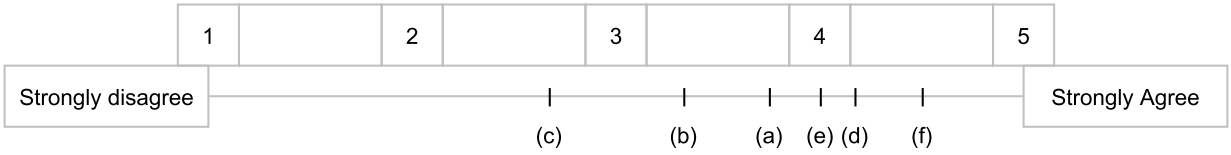

Figure 14: Average agreement values: (a) Multiplicities first example. (b) Multiplicities second example. (c) Order of attributes in a class. (d) Solution in regard to the merging of a `String/Integer` data type. (e) Merging of inconsistent role names. (f) No calculation of multiplicities for contradictory associations.

so the first part of research question 2 can be answered mostly positive. However, there were different patterns of behavior for some of the more complex tasks. The greatest disparity happened for the multiplicities. The most common approach was to just label every multiplicity as `0..*`, sometimes participants also used a `1..*` multiplicity. Sometimes, participants left association names and role names empty. To answer the final part of the research question, the acceptance ratings from the tasks for the tool have to be observed. Figure 14 shows the average agreement values. The agreement in regard to the multiplicities was (a) for the first example 3.75/5 and (b) for the second example 3.33/5. While that is not a terrible agreement score, some enhancements have to be made in regard to the multiplicities. The agreement for the order of the attributes in a class (c) was even lower with 2.67/5, so there will be enhancements in future work. The solution in regard to the merging of a `String/Integer` data type (d) was better accepted with a score of 4.17/5. The merging of the inconsistent role names (e) was accepted with a score of 4/5 and that the program does not calculate multiplicities for contradictory associations (f) was scored 4.5/5. To sum up, the answer to research question 2 is a mixed one. A lot of the patterns are the same as the ones proposed in this paper, others are not. Some of the differences are accepted, while others are not.

5.3.3 Research question 3: Research question 3 was answered with the help of the System Usability Scale (SUS) [3], which is a widespread tool to review usability. It consists of 10 questions and the result is a value between 0 and 100. The only change to the SUS that was done, was the change proposed in [5], which should help non-native speakers to understand one of the questions.

The result of the SUS for this evaluation was an average of 75.7 out of 100. This is a good score when compared to other results, which was found out to be an average of 68 across 500 different evaluations [18]. This means the current version is already in an acceptable state.

5.4 Lessons Learned and Future Evaluations

This evaluation was a success in a way that not only expected problems were detected, but also new ideas for concepts not yet thought of. For example, the lowest agreement value of the participants was on the order of the attributes in the classes after the transformation. This non-trivial problem could be one of many approaches for future work on this topic.

Currently, the tool is also used for teaching purposes in a modeling course[6]. As part of an ongoing evaluation, the results and feedback from the students will be used to further improve the tool and the transformation in general.

What could be an important research question for future evaluations would be to ask for more details about the usefulness of the idea. In which respect is this approach of moving from the informal to the formal effective? In other words, for which criteria does it actually help in the design process (e.g., increases productivity or quality of designs)?

[6] http://www.db.informatik.uni-bremen.de/teaching/courses/ss2018_eis/

6 RELATED WORK

We see the related work roughly in three categories. First we look at example based modeling which modeling based on object diagrams falls into. Then we look at transformations in the modeling field. Finally and maybe most importantly, we look into the work that was done in the area of uncertainty in modeling.

6.1 Example Based Modeling

López et al. [10] followed an approach where domain experts, who are not so familiar with modeling, and modeling experts come together to create a metamodel. The domain experts create model fragments which get absorbed by the metamodel, in part automatically and in part with the supervision of the modeling experts. In a next step, the domain experts can validate the created metamodel with the help of examples.

Maoz et al. [11] explain a very similar approach, that also looks at object diagrams and class diagrams. The object diagrams are extended to allow for positive/negative and example/invariant modalities. The class diagrams, on the other hand, are created manually and later verified with the help of the modal object diagrams.

Zayan et al. [22] also use UML class and object diagrams. The object diagrams can represent positive and negative examples. However, the focus lies more on model comprehension and uses object diagrams as extensions to help domain experts to understand the more abstract class diagrams.

6.2 Transformations

Kappel et al. [9] give an overview of the work that was done in the model-transformation-by-example field. Instead of working with the computer internal representation, the models are described by concrete examples. The work is focused on transformations where the input and output is equivalent, opposed to the approach in this paper, where input and output have different meanings.

Mens and Van Gorp [12] look at model transformations in general and put them into groups. For example, they distinguish between horizontal transformations (staying on the same abstraction level) and vertical transformations (going to a different abstraction level). Using this distinction, the transformation from this paper would be a vertical one. They also mention the need to have mechanisms for inconsistency management when dealing with incomplete or inconsistent models.

Smid and Rensink [21] use the tool GROOVE (GRaphs for Object-Oriented VErification) to restructure class diagrams. This solution is comparable to this paper in a way, that it uses a graphical tool which allows for visual feedback for the transformation. However, the described transformation is a horizontal one.

6.3 Uncertainty in Modeling

Salay et al. [16] focused on formalizing informal notations. They talk about how sometimes information is just not complete at the time of model creation and how this incomplete information can be incorporated anyway. These additions are formalized and verified in a theoretical way.

Semeráth and Varró [20] also talk about partial or incomplete models. In their work, they stay on the same abstraction level, meaning that they look at every possible model that may come

from such a partial model. Their evaluation focused on efficiency, opposed to the approach and usability study done for this paper.

Famelis and Santosa [4] already proposed a notation for design uncertainty which builds on the previous work of Salay, Famelis, and Chechik regarding partial models [17].

6.4 Relation to this Contribution

What distinguishes all of those works from this contribution, is the notation in the form of placeholders and other markers in combination with a strictly automated transformation. This combination allows not only to show which parts are missing or contradictory, but also allows fixing these problems iteratively. In some of the mentioned works, modeling experts create corresponding class diagrams by hand while in our approach, the tool creates the class diagram automatically. A final distinction is the evaluation that was done with potential users of the transformation that also kept in mind the usability of our tool. In the end, the tool should help with modeling and not frustrate users with poor controls.

7 CONCLUSION AND FUTURE WORK

The problem that has been tackled here was how to support software developers in freely expressing their ideas during (model-based) development. This general problem was discussed in a focused context and by means of a transformation from sketchy object diagrams to improvable class diagrams. We have introduced the notions "Complete" and "Incomplete" for object diagram elements and additionally "Contradictory" for class diagram elements. The approach was implemented as a plug-in for a modeling tool. In order to validate the underlying concepts and the implementation, a successful evaluation was undertaken with developers familiar with object-oriented modeling. The evaluation supported our thesis that starting development from informal sketches and ending up in formal models was perceived positively by developers. It revealed expected problems, but also brought up new ideas for problems not yet thought of.

A future task is the treatment of multiplicities. It will be no difficulty to implement an option in the class diagram to choose from (a) the currently realized exact multiplicities from the object diagram, (b) the most frequently used multiplicities `0..1`, `1`, `1..*`, `0..*` and (c) developer specified multiplicities. Additional UML class diagram concepts to be implemented include inheritance, higher-order associations, and part-whole relationships. New markers will be introduced on the class side that emphasize where guesses were made during the transformation and how well-founded they are. Markers will be used to highlight minor inconsistencies like typos or guesses like when the order of association ends is involved.

As OCL expertise is available in our context, a possible direction could be to support developers in designing OCL expressions in a liberal way, i.e., in the sense that OCL expressions need not necessarily follow exact OCL syntax. Markers could indicate spots for correction and improvement, and from imperfect OCL expressions the maximal part still in line with "formal" OCL could be extracted.

In the span between class diagrams and object diagrams there is another category: collaboration diagrams. They have the advantage of being both general and yet still instance-oriented. Using collaborations in addition to object models is an interesting option. Collaborations have the advantage that they are more general than objects and more in line with true object-oriented design than either class models or object models. Last but not least, larger case studies in the academic, industrial and governmental context must check the applicability of the proposed approach. Supporting developers in freely expressing their ideas is a wide field.

REFERENCES

[1] David Barnes and Michael Kölling. 2009. *Objects First with Java.* Pearson.
[2] Grady Booch, James Rumbaugh, and Ivar Jacobson. 1999. *The Unified Modeling Language User Guide.* Addison-Wesley.
[3] John Brooke. 1996. SUS-A Quick and Dirty Usability Scale. *Usability Evaluation in Industry* (1996), 189–194.
[4] Michalis Famelis and Stephanie Santosa. 2013. MAV-Vis: A notation for model uncertainty. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. 7–12.
[5] Kraig Finstad. 2006. The System Usability Scale and Non-Native English Speakers. *Journal of Usability Studies* 1, 4 (2006), 185–188.
[6] Martin Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming* 69 (2007), 27–34.
[7] Martin Gogolla, Frank Hilken, and Khanh-Hoang Doan. 2017. Achieving Model Quality through Model Validation, Verification and Exploration. *Journal on Computer Languages, Systems and Structures, Elsevier, NL* (2017). Online 2017-12-02.
[8] Martin Gogolla, Frank Hilken, and Andreas Kästner. 2018. Some Narrow and Broad Challenges in MDD. In *Software Technologies: Applications and Foundations*, Martina Seidl and Steffen Zschaler (Eds.). Springer International Publishing, Cham, 172–177.
[9] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. 2012. Model Transformation By-Example: A Survey of the First Wave. In *Conceptual Modelling and Its Theoretical Foundations*, Antje Düsterhöft, Meike Klettke, and Klaus-Dieter Schewe (Eds.). Springer Berlin Heidelberg, 197–215.
[10] Jesús J López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2015. Example-driven meta-model development. *Software & Systems Modeling* 14, 4 (2015), 1323–1347.
[11] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. Modal Object Diagrams. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, 281–305.
[12] Tom Mens and Pieter Van Gorp. 2006. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science* 152 (2006), 125–142.
[13] Nikolaus Regnat. 2018. Why SysML does often fail and possible solutions. In *Modellierung 2018*, Ina Schaefer, Dimitris Karagiannis, Andreas Vogelsang, Daniel MÃĺndez, and Christoph Seidl (Eds.). Gesellschaft für Informatik e.V., Bonn, 17–20.
[14] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2005. *The Unified Modeling Language Reference Manual* (2nd ed.). Addison-Wesley.
[15] Bernhard Rumpe. 2004. *Modellierung mit UML.* Springer.
[16] Rick Salay, Marsha Chechik, Michalis Famelis, and Jan Gorzny. 2015. A Methodology for Verifying Refinements of Partial Models. *Journal of Object Technology* 14, 3 (2015). https://doi.org/10.5381/jot.2015.14.3.a3
[17] Rick Salay, Michalis Famelis, and Marsha Chechik. 2012. Language Independent Refinement Using Partial Modeling. In *Fundamental Approaches to Software Engineering*, Juan de Lara and Andrea Zisman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 224–239.
[18] Jeff Sauro. 2011. *A Practical Guide to the System Usability Scale: Background, Benchmarks & Best Practices.* Measuring Usability LLC.
[19] Bran Selic. 2016. Career Award Talk. https://youtu.be/9qPbGksB3d4?t=20m32s. (2016).
[20] Oszkár Semeráth and Dániel Varró. 2017. Graph Constraint Evaluation over Partial Models by Constraint Rewriting. In *Theory and Practice of Model Transformation*, Esther Guerra and Mark van den Brand (Eds.). Springer International Publishing, Cham, 138–154.
[21] Wietse Smid and Arend Rensink. 2013. Class Diagram Restructuring with GROOVE. In *Proceedings Sixth Transformation Tool Contest (Electronic Proceedings in Theoretical Computer Science)*, P. Van Gorp, L.M. Rose, and C. Krause (Eds.). 83–87.
[22] Dina Zayan, Atrisha Sarkar, Michał Antkiewicz, Rita Suzana Pitangueira Maciel, and Krzysztof Czarnecki. 2018. Example-driven modeling: on effects of using examples on structural model comprehension, what makes them useful, and how to create them. *Software & Systems Modeling* (2018).